CORE JAVA

Akshita Chanchlani

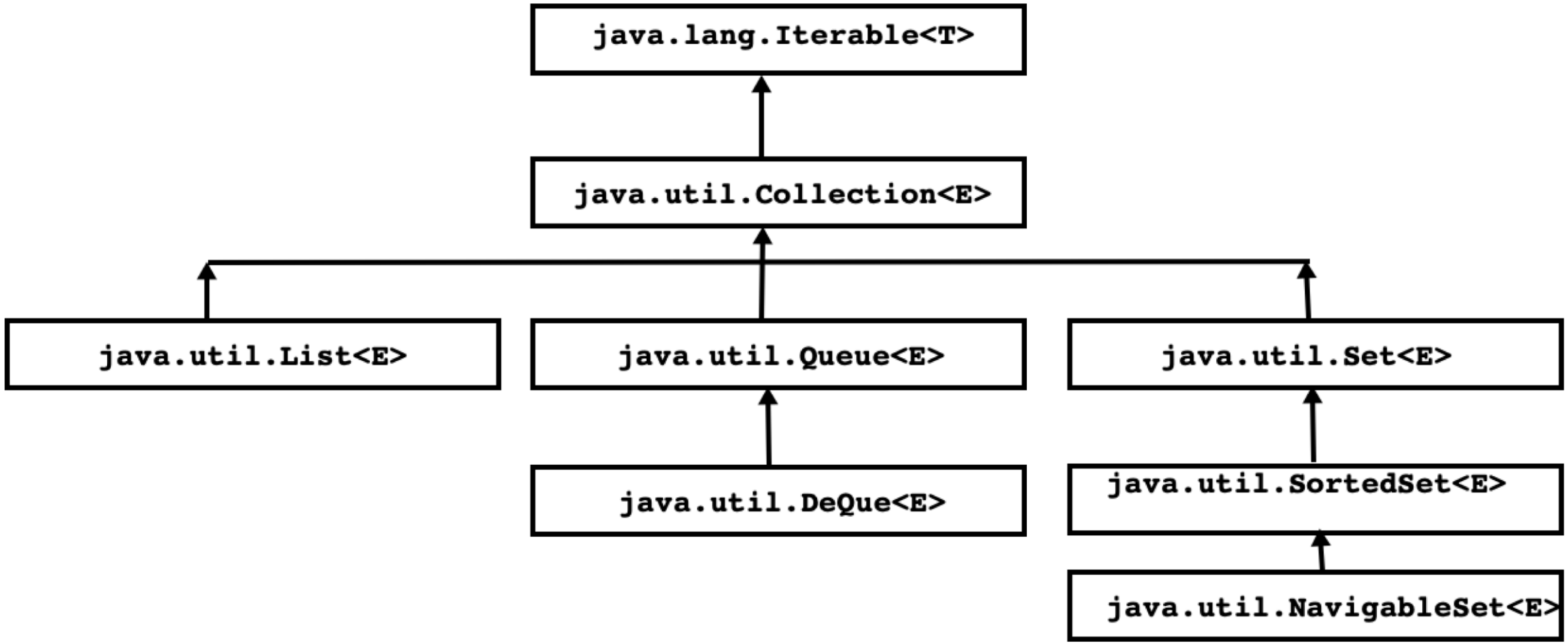
Collection Framework

- Every value/data stored in data structure is called element.
- A group of data (Object reference) handled as a single unit.
- Framework is library of reusable classes/interfaces that is used to develop application.
- **Library of reusable data structure classes that is used to develop java application is called collection framework.**
- Main purpose of collection framework is to manage data in RAM efficiently.
- Consider following Example:
    1. Person has-a birthdate
    2. Employee is a person
- In java, collection instance do not contain instances rather it contains reference of instances.
- If we want to use collection framework them we should **import java.util** package.

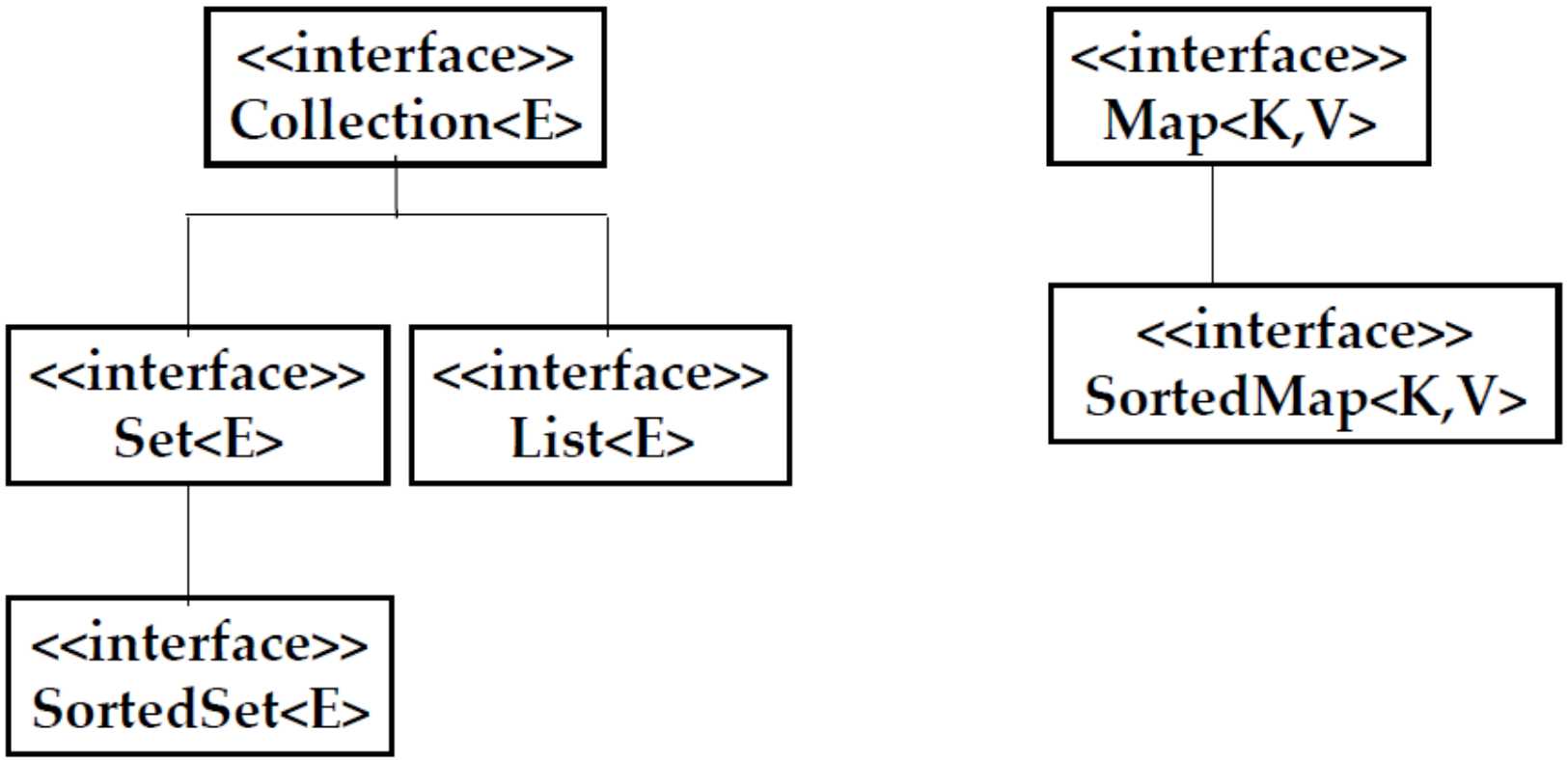
Collection Framework

- Collection frameworks encapsulates the algorithms
- Programmer don't have to worry abut the implementation of the various collections types.
- Nevertheless, still collection framework is encourages programmer to extend members of collection to create a specific implementation where one is needed

Collection Interface Hierarchy



Collection Framework Interface Hierarchy



Collection framework is in java.util package

Iterable<T>

- It is a interface declared in java.lang package.

- All the collection classes implements Iterable interface hence we can traverse it using for each loop

- Methods of Iterable interface:
    1. Iterator<T> **iterator**()
    2. default Spliterator<T> **spliterator**()
    3. default void **forEach**(Consumer<? super T> action)

Collection<E>

- Collection<E> is interface declared in java.util package.
- It is sub interface of Iterable interface.
- It is **root interface** in collection framework interface hierarchy.
- Default methods of Collection interface
  1. default Stream<E> **stream**()
  2. default Stream<E> **parallelStream**()
  3. default boolean **removeIf**(Predicate<? super E> filter)

Collection<E>

- Abstract Methods of Collection Interface

```
1. boolean add(E e)
2. boolean addAll(Collection<? extends E> c)
3. void clear()
4. boolean contains(Object o)
5. boolean containsAll(Collection<?> c)
6. boolean isEmpty()
7. boolean remove(Object o)
8. boolean removeAll(Collection<?> c)
9. boolean retainAll(Collection<?> c)
10.int size()
11.Object[] toArray()
12.<T> T[] toArray(T[] a)
```

There is no direct concrete implementation class for Collection interface but exists for sub interface

This interface supports the most basic and general operations and queries that can be performed on a group of objects.

List<E>

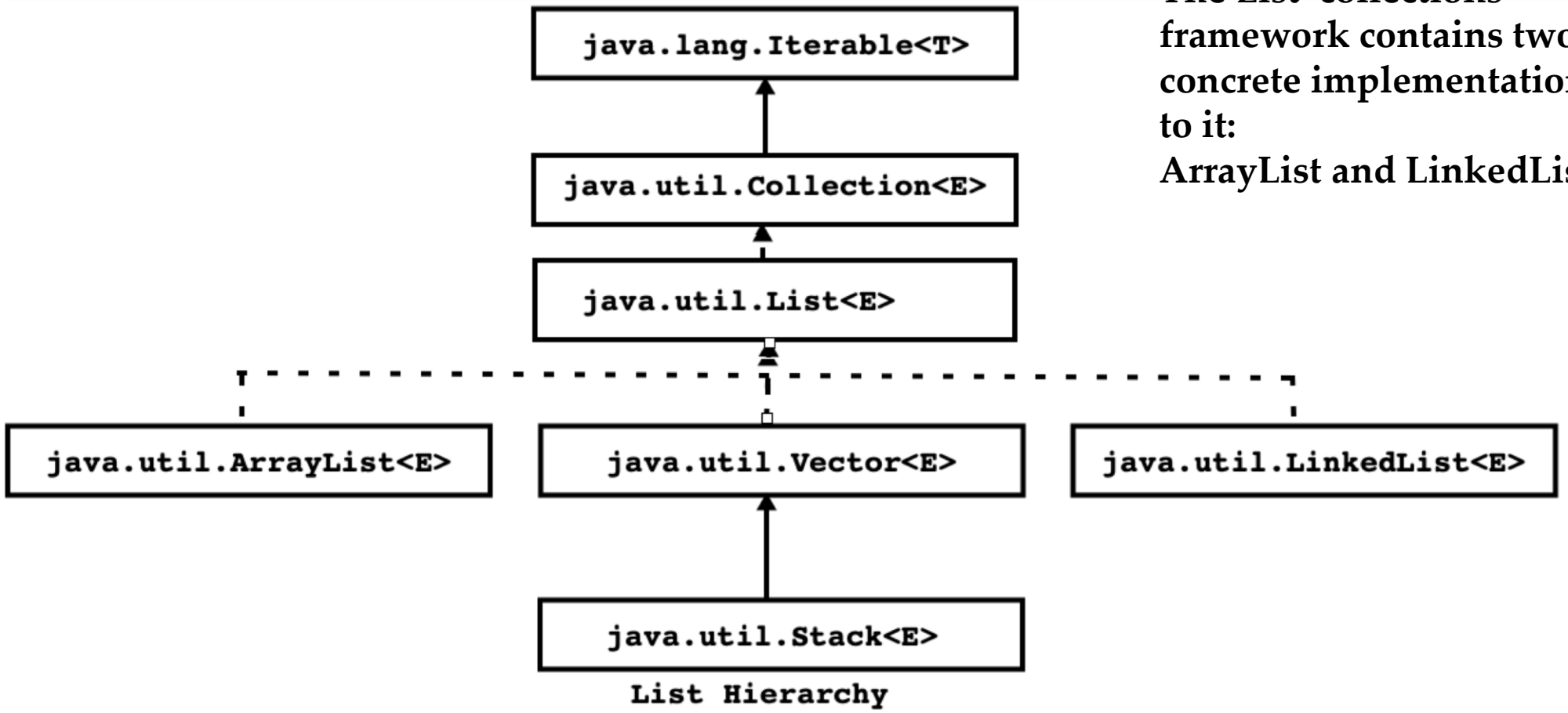
- It is sub interface of java.util.Collection interface.
- It is ordered/sequential collection.
- **ArrayList, Vector, Stack, LinkedList etc. implements List interface. It generally referred as "List collections".**
- List collection can contain duplicate element as well multiple elements.
- Using integer index, we can access elements from List collection.
- We can traverse elements of List collection using Iterator as well as ListIterator.
- A list is a collection in which duplicate elements are allowed, and where the order is significant null
- List interface inherits from Collection, but changes the semantics of some methods, and adds new methods. The list starts its index at 0.

<a,b,c>, <c,a,b>and <a,b,b,c>are different lists

List<E>

- Abstract methods of List Interface
  1. void **add**(int index, E element)
  2. boolean **addAll**(int index, Collection<? extends E> c)
  3. E **get**(int index)
  4. int **indexOf**(Object o)
  5. int **lastIndexOf**(Object o)
  6. ListIterator<E> **listIterator**()
  7. ListIterator<E> **listIterator**(int index)
  8. E **remove**(int index)
  9. E **set**(int index, E element)
  10. List<E> **subList**(int fromIndex, int toIndex)

List Interface Hierarchy



List Hierarchy

The List collections framework contains two concrete implementations to it:
ArrayList and LinkedList.

ArrayList<E>

- It is resizable array.
- It implements List<E>, RandomAccess, Cloneable, Serializable interfaces.
- It is List collection.
- It is unsynchronized collection. Using"Collections.synchronizedList" method, we can make it synchronized.
  - List list = Collections.synchronizedList(new ArrayList(...));
- Initial capacity of ArrayList is 10. If ArrayList is full then its capacity gets increased by half of its existing capacity.
- ArrayList is recommended when you're adding and removing elements only to the end of the collection satisfies you and when you wish to access elements using their indices.
- It is less time-consuming than LinkedList is.

Vector<E>

- It is resizable array.
- It implements List<E>, RandomAccess, Cloneable, Serializable.
- It is List collection.
- It is synchronized collection.
- Default capacity of vector is 10. If vector is full then its capacity gets increased by its existing capacity.
- We can traverse elements of vector using Iterator, ListIterator as well as Enumeration.
- **Note:** If we want to manage elements of non final type inside Vector then non final type should override "equals" method.
- It is also called as growable array.

Vector<E>

- Duplicate elements are allowed in the vector class.
- It preserves the insertion order in Java.
- Null elements are allowed in the Java vector class.
- Heterogeneous elements are allowed in the vector class. Therefore, it can hold elements of any type and any number.
- Most of the methods present in the vector class are synchronized. That means it is a thread-safe. Two threads cannot access the same vector object at the same time. Only one thread can access can enter to access vector object at a time.
- Vector class is preferred where we are developing a multi-threaded application but it gives poor performance because it is thread-safety.
- Vector is rarely used in a non-multithreaded environment due to synchronized which gives you poor performance in searching, adding, delete, and update of its element.
- It can be iterated by a simple for loop, Iterator, ListIterator, and Enumeration.
- Vector is the best choice if the frequent operation is retrieval (getting).

Synchronized Collections

- 1. **Vector**
- 2. **Stack**(Sub class of Vector)
- 3. **Hashtable**
- 4. **Properties**(Sub class of Hashtable)

Enumeration<E>

- It is interface declared in java.util package.
- Methods of Enumeration I/F
    1. boolean hasMoreElements()
    2. E nextElement()
- It is used to traverse collection only in forward direction. During traversing, we can add, set or remove element from collection.
- It is introduced in jdk 1.0.
- "public Enumeration<E> elements()" is a method of Vector class.

```
Integer element = null;
Enumeration<Integer> e = v.elements();
while( e.hasMoreElements()){
    element = e.nextElement();
    System.out.println(element);
}
```

Iterator<E>

- It is a interface declared in java.util package.
- It is used to traverse collection only in forward direction. During traversing, we can not add or set element but we can remove element from collection.
- Methods of Iterator
  1. boolean hasNext()
  2. E next()
  3. default void remove()
  4. default void forEachRemaining(Consumer<? super E> action)
- It is introduced in jdk 1.2

- **Iterator takes the place of Enumeration in the Java collections framework.**
- **Iterators allow the caller to remove elements from the underlying collection during the iteration.**

```
Integer element = null;
Iterator<Integer> itr = v.iterator();
while( itr.hasNext()){
    element = itr.next();
    System.out.  println(element);
}
```

ListIterator<E>

- It is sub interface of Iterator interface.
- It is used to traverse only List Collection in bidirectional.
- During traversing, we can add, set as well as remove element from collection.
- It is introduced in jdk 1.2
- Methods of ListIterator
    1. boolean hasNext()
    2. E next()
    3. boolean hasPrevious()
    4. E previous()
    5. void add(E e)
    6. void set(E e)
    7. void remove()

ListIterator<E>

```
Integer element = null;
ListIterator<Integer> itr = v.listIterator();
while( itr.hasNext()){
    element = itr.next();
    System.out.print(element+"   ");
}

while( itr.hasPrevious()){
    element = itr.previous();
    System.out.print(element+"   ");
}
```

Types of Iterator

- **Fail Fast Iterator**
- During traversing, using collection reference, if we try to modify state of collection and if iterator do not allows us to do the same then such iterator is called "Fail Fast" Iterator. In this case JVM throws **ConcurrentModificationException.**

```
Integer element = null;
Iterator<Integer> itr = v.iterator();
while( itr.hasNext()){
    element = itr.next();
    System.out.print(element+"   ");
    if( element == 50 )
        v.add(60); //ConcurrentModificationException
}
```

Types of Iterator

- **Fail Safe Iterator**
- During traversing, if iterator allows us to do changes in underlying collection then such iterator is called fail safe iterator.

```
Integer element = null;
Enumeration<Integer> e = v.elements();
while( e.hasMoreElements()){
    element = e.nextElement();
    System.out.print(element+" ");
    if( element == 50 )
        v.add(60); //OK
}
```

Stack<E>

- It is linear data structure which is used to manage elements in Last In First Out order.
- It is sub class of Vector class.
- It is synchronized collection.
- It is List Collection.
- Methods of Stack class
    1. public boolean empty()
    2. public E push(E item)
    3. public E peek()
    4. public E pop()
    5. public int search(Object o)
- * Since it is synchronized collection, it slower in performance.
- * For high performance we should use ArrayDeque class.

LinkedList<E>

- It is a List collection.
- It implements List<E>, Deque<E>, Cloneable and Serializable interface.
- Its implementation is depends on Doubly linked list.
- It is unsynchronized collection. Using Collections.synchronizedList() method, we can make it synchronized.
  - **List list = Collections.synchronizedList(new LinkedList(...));**
- Note : If we want to manage elements of non-final type inside LinkedList then non final type should override "equals" method.
- Instantiation
  - **List<Integer> list = new LinkedList<>();**

LinkedList is best when add and remove operations happen anywhere, not only at the end.

LinkedList<E>

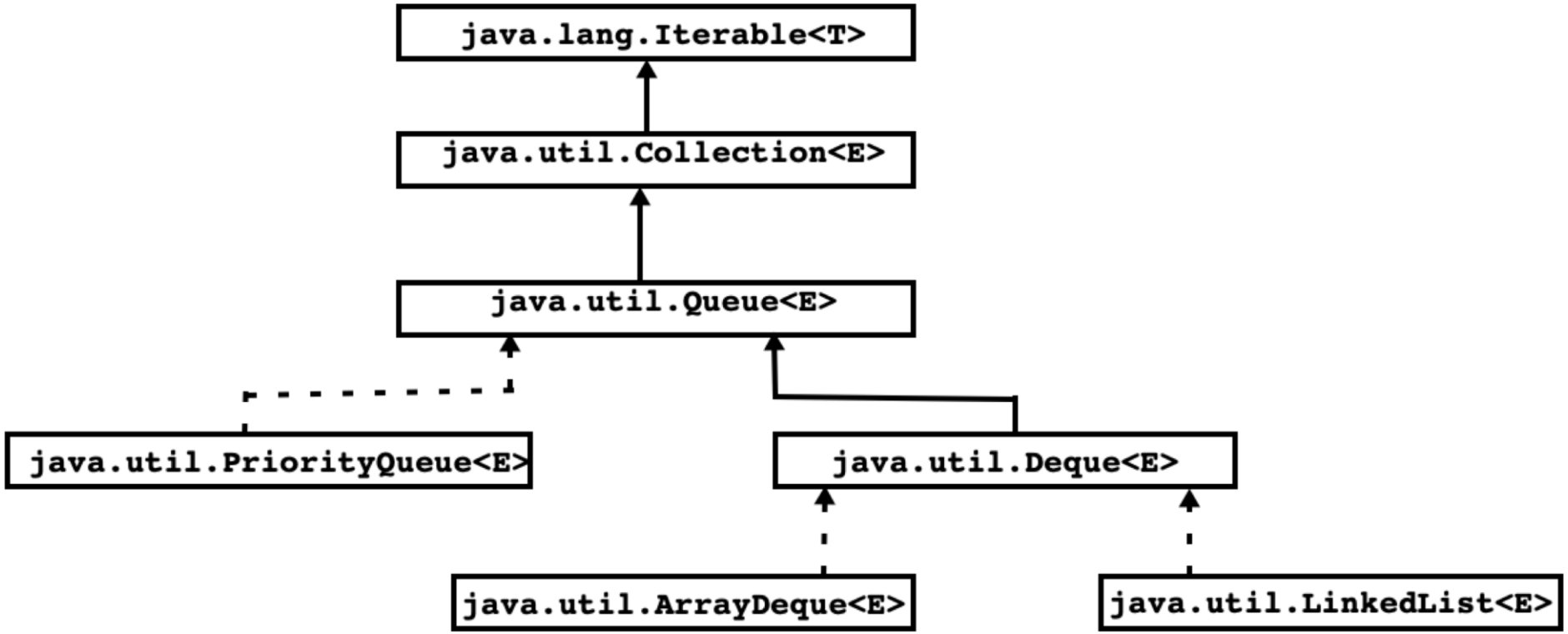
Arrays have certain limitations such as:

- Size of the array is fixed
- Array elements need contiguous memory locations
- Inserting an element in an array is performance wise expensive
- deleting an element from the array is also a performance wise expensive

Linked List by providing following features:

- Allows **dynamic memory allocation**
- elements **don't need contiguous memory locations**
- Insert and delete operations in the Linked list are not performance wise expensive because adding and deleting an element from the linked list does't require element shifting, only the pointer of the previous and the next node requires change.

Queue Interface Hierarchy



Queue Hierarchy

Queue<E>

- It is interface declared in java.util package.
- It is sub interface of Collection interface.
- It is introduced in jdk 1.5

Summary of Queue methods

	Throws exception	*Returns special value*
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

Since the Queue is an interface, we cannot provide the direct implementation of it.

Classes that implement Queue Interface in Java:

1.Priority Queue

2.Dequeue

Queue Interface Methods

add()

- Inserts the specified element into the queue. If the task is successful, add() returns true, if not it throws an exception.

offer()

- Inserts the specified element into the queue. If the task is successful, offer() returns true, if not it returns false.

element()

- Returns the head of the queue. Throws an exception if the queue is empty.

peek()

- Returns the head of the queue. Returns null if the queue is empty.

remove()

- Returns and removes the head of the queue. Throws an exception if the queue is empty.

poll()

- Returns and removes the head of the queue. Returns null if the queue is empty.

Queue<E>

```java
Queue<Integer> que = new ArrayDeque<>();
que.offer(10);
que.offer(20);
que.offer(30);

Integer ele = null;
while( !que.isEmpty()){
    ele = que.peek();
    //TODO : processing
    que.poll();
}
```

```java
Queue<Integer> que = new ArrayDeque<>();
que.add(10);
que.add(20);
que.add(30);
Integer ele = null;
while( !que.isEmpty()){
    ele = que.element();
    //TODO : Processing
    que.remove();
}
```

Deque<E>

- It is usually pronounced "deck".
- It is sub interface of Queue.
- It is introduced in jdk 1.6
- If we want to perform operations from bidirection then we should use Deque interface.

Summary of Deque methods

	First Element (Head)		Last Element (Tail)	
	Throws exception	*Special value*	*Throws exception*	*Special value*
Insert	`addFirst(e)`	`offerFirst(e)`	`addLast(e)`	`offerLast(e)`
Remove	`removeFirst()`	`pollFirst()`	`removeLast()`	`pollLast()`
Examine	`getFirst()`	`peekFirst()`	`getLast()`	`peekLast()`

Deque<E>

- Deque Interface is a linear collection that supports element insertion and removal at both ends.
- The class which implements this interface is ArrayDeque.
- It extends the Queue interface.
- Deque is an interface and has two implementations: LinkedList and ArrayDeque.
- **Creating a Deque**
  Syntax : Deque dq = new LinkedList();
  Deque dq = new ArrayDeque();

ArrayDeque

ArrayDeque class provides the facility of using deque and resizable-array.
It inherits the AbstractCollection class and implements the Deque interface.

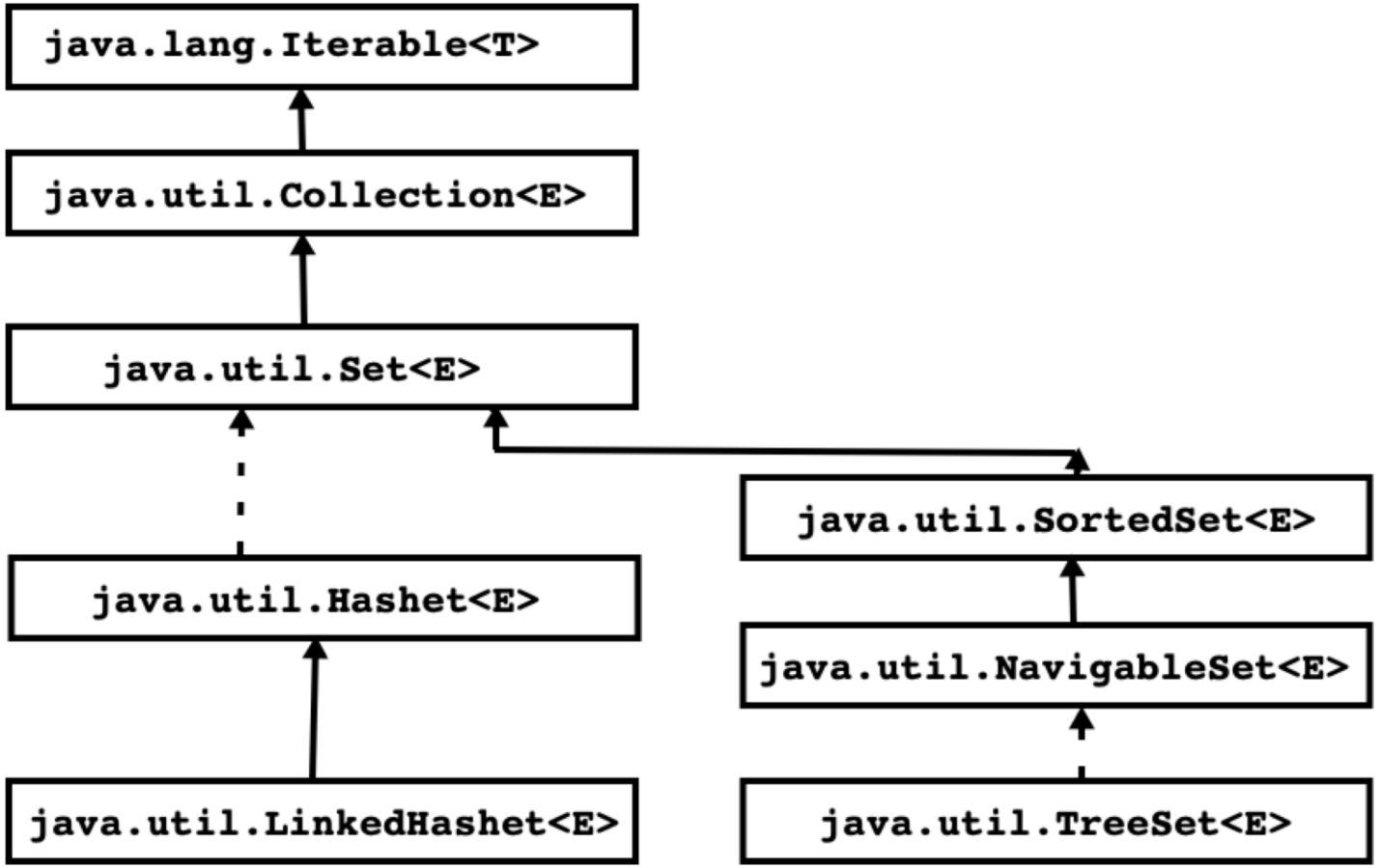
Syntax : Deque<String> arr = new ArrayDeque<String>();

Priority Queue

- PriorityQueue class provides the functionality of the heap data structure.
- The PriorityQueue class provides the facility of using a queue.
- It does not order the elements in a FIFO manner.
- It is based on Priority Heap.
- The elements of the priority queue are ordered according to the natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.

Syntax : **PriorityQueue<Integer> numbers = new PriorityQueue<Integer>();**
// it will create PriorityQueue without any arguments
//head of the queue will be smallest element of the queue
//elements are removed in ascending order from the queue.

Set Interface Hierarchy



Set<E>

- It is sub interface of java.util.Collection interface.
- HashSet, LinkedHashSet, TreeSet etc. implements Set interface. It is also called as Set collection.
- Set collections do not contain duplicate elements.
- Set will not maintain any order for elements
- While adding new element it is using Object's equals() and hashCode() method to check , if such element is there or not in set

There are three concrete Set implementations that are part of the Collection Framework: HashSet, TreeSet, and LinkedHashSet.

HashSet and TreeSet, LinkedHashSet

- You use HashSet, which maintains its collection in an unordered manner.
- If this doesn't suit your needs, you can use TreeSet.
- A TreeSet keeps the elements in the collection in sorted order
- While HashSet has an undefined order for its elements, LinkedHashSet supports iterating through its elements in the order they were inserted.
- Understand that the additional features provided by TreeSet and LinkedHashSet add to the runtime costs.

TreeSet<E>

- It is Set collection.
- It can not contain duplicate element as well as null element.
- It is sorted collection.
- Its implementation is based on TreeMap
- It is unsynchronized collection.
- Using "Collections.synchronizedSortedSet()" method we can make it synchronized.
  - **SortedSet s = Collections.synchronizedSortedSet(new TreeSet(...));**
- It is introduced in jdk 1.2
- Note : If we want to manage elements of non final type inside TreeSet then non final type should implement Comparable interface.
- Instantiation
  - **Set<Integer> set = new TreeSet<>( );**

Hashing

- Hashing is a searching algorithm which is used to search element in constant time(faster searching).
- In case array, if we know index of element then we can locate it very fast.
- Hashing technique is based on "hashcode".
- Hashcode is not a reference or address of the object rather it is a logical integer number that can be generated by processing state of the object.
- Generating hashcode is a job of hash function/method.
- Generally hashcode is generated using prime number.

```
//Hash Method
private static int getHashCode(int data)
{
    int result = 1;
    final int PRIME = 31;
    result = result * data + PRIME * data;
    return result;
}
```

Hashing

- If state of object/instance is same then we will get same hashcode.
- Hashcode is required to generate slot.
- If state of objects are same then their hashcode and slot will be same.
- By processing state of two different object's , if we get same slot then it is called collision.
- Collision resolution techniques:
  - Seperate Chaining / Open Hashing
  - Open Addressing / Close Hashing
    1. Linear Probing
    2. Quadratic Probing
    3. Double Hashing / Rehashing

Hashing

- Collection(LinkedList/Tree) maintained per slot is called bucket.
- Load Factor = (Count of bucket / Total elements);
- **In hashcode based collection, if we want manage elements of non final type then reference type should override equals() and hashcode() method.**
- hashCode() is non final method of java.lang.Object class.
- Syntax:
  - public native int hashCode( );
- On the basis of state of the object, we want to generate hashcode then we should override hashCode() method in sub class.
- The hashCode method defined by class Object does return distinct integers for distinct objects. This is typically implemented by converting the internal address of the object into an integer.

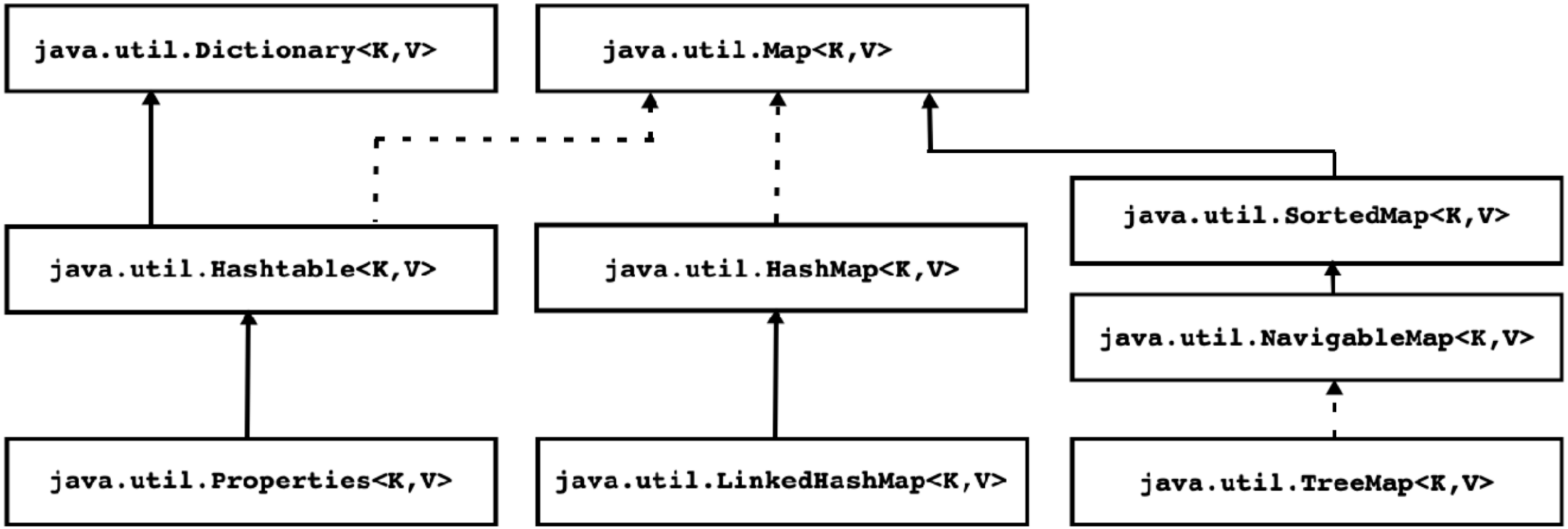
HashSet<E>

- It Set Collection.
- It can not contain duplicate elements but it can contain null element.
- It's implementation is based on HashTable.
- It is unordered collection.
- It is unsynchronized collection. Using Collections.synchronizedSet() method, we can make it synchronized.
- It is introduced in jdk 1.2
- Note : If we want to manage elements of non final type inside HashSet then non final type should override equals and hashCode() method.
- Instantiation:
  - **`Set<Integer> set = new HashSet<>();`**

LinkedHashSet<E>

- It is sub class of HashSet class.
- Its implementation is based on linked list and Hashtable.
- It is ordered collection.
- It is unsynchronized collection. Using Collections.synchronizedSet() method we can make it synchronized.
  - **Set s = Collections.synchronizedSet(new LinkedHashSet(...));**
- It is introduced in jdk 1.4
- It can not contain duplicate element but it can contain null element.

Map Interface Hierarchy



Dictionary<K,V>

- It is abstract class declared in java.util package.
- It is super class of Hashtable.
- It is used to store data in key/value pair format.
- It is not a part of collection framework
- It is introduced in jdk 1.0
- Methods:
    1. public abstract boolean isEmpty()
    2. public abstract V put(K key, V value)
    3. public abstract int size()
    4. public abstract V get(Object key)
    5. public abstract V remove(Object key)
    6. public abstract Enumeration<K> keys()
    7. public abstract Enumeration<V> elements()
- Implementation of Dictionary is Obsolete.

Map<K,V>

- It is part of collection framework but it doesn't extend Collection interface.
- This interface takes the place of the Dictionary class, which was a totally abstract class rather than an interface.
- HashMap, Hashtable, TreeMap etc are Map collection's.
- Map collection stores data in key/value pair format.
- In map we can not insert duplicate keys but we can insert duplicate values.
- It is introduced in jdk 1.2
- Map.Entry<K,V> is nested interface of Map<K,V>.
- Following are abstract methods of Map.Entry interface.
    1. K getKey()
    2. V getValue()
    3. V setValue(V value)

Map<K,V>

- Abstract Methods of Map<K,V>
  1. boolean isEmpty()
  2. V put(K key, V value)
  3. void putAll(Map<? extends K,? extends V> m)
  4. int size()
  5. boolean containsKey(Object key)
  6. boolean containsValue(Object value)
  7. V get(Object key)
  8. V remove(Object key)
  9. void clear()
  10. Set<K> keySet()
  11. Collection<V> values()
  12. Set<Map.Entry<K,V>> entrySet()

Map Implementation
Collection framework gives two classes
HashMap and TreeMap

- An instance, whose type implements Map.Entry<K,V> interface is called enrty instance.

HashMap , TreeMap

- By default, choose HashMap, it serves the most needs.
- TreeMap implementation will maintain the keys of the map in a sorted order.
- it's better to simply keep everything in a HashMap while adding, and create a TreeMap at the end:
- Map <String,String> map = new HashMap<String,String>();// Add and remove elements from unsorted mapmap.put("Foo", "Bar");map.put("Bar", "Foo");map.remove("Foo");map.put("Foo", "Baz");// Then sort before displaying elements// in sorted ordermap = new TreeMap(map);

Hashtable<K,V>

- It is Map<K,V> collection which extends Dictionary class.
- It can not contain duplicate keys but it can contain duplicate values.
- In Hashtable, Key and value can not be null.
- It is synchronized collection.
- It is introduced in jdk 1.0
- In Hashtable, if we want to use instance non final type as key then it should override equals and hashCode method.

HashMap<K,V>

- It is map collection
- It's implementation is based on Hashtable.
- It can not contain duplicate keys but it can contain duplicate values.
- In HashMap, key and value can be null.
- It is unsynchronized collection. Using Collections.synchronizedMap() method, we can make it synchronized.
  - `Map m = Collections.synchronizedMap(new HashMap(...));`
- It is introduced in jdk 1.2.
- Instantiation
  - `Map<Integer, String> map = new HashMap<>( );`
- **Note : In HashMap, if we want to use element of non final type as a key then it should override equals() and hashCode() method.**

LinkedHashMap<K,V>

- It is sub class of HashMap<K,V> class
- Its implementation is based on LinkedList and Hashtable.
- It is Map collection hence it can not contain duplicate keys but it can contain duplicate values.
- In LinkedHashMap, key and value can be null.
- It is unsynchronized collection. Using Collections.synchronizedMap() method we can make it synchronized.
  - **Map m = Collections.synchronizedMap(new LinkedHashMap(...));**
- LinkedHashMap maintains order of entries according to the key.
- Instantiation:
  - **Map<Integer, String> map = new LinkedHashMap<>();**
- It is introduced in jdk 1.4

TreeMap<K,V>

- It is map collection.
- It can not contain duplicate keys but it can contain duplicate values.
- It TreeMap, key not be null but value can be null.
- Implementation of TreeMap is based on Red-Black Tree.
- It maintains entries in sorted form according to the key.
- It is unsynchronized collection. Using Collections.synchronizedSortedMap() method, we can make it synchronized.
  - **SortedMap m = Collections.synchronizedSortedMap(new TreeMap(...));**
- Instantiation:
  - **Map<Integer, String> map = new TreeMap<>();**
- It is introduced in jdk 1.2
- Note : In TreeMap, if we want to use element of non final type as a key then it should implement Comparable interface.

Thank You.